M12-L1 Problem 2

Sometimes the dimensionality is greater than the number of samples. For example,in this problem X has 19 features, but there are only 4 data points. You will need to use the alternate PCA formulation in this case. Follow the steps in the cells below to implement this method.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

X = np.array([  [-2,  1,  2, -3,  4,  1,  0,  3,  0,  2,  1,  1,  2,  3, -2, -3, 2,  1,  0],
                [ 1,  2, -4,  2, -4,  2,  5,  2,  2,  1, -3,  0,  0,  1, -2,  1, 1, -3, -2],
                [ 1, -3,  2,  1,  0, -3, -5, -1,  3,  3, -2, -3, -2, -1,  1,  0, 5,  4,  2],
                [ 3, -1,  0,  2,  2, -5, -4, -1,  2, -1,  3,  4,  4,  2,  1,  2, -2,  1, -1]])
```

Computing Principal Components

The A matrix

First, you should compute the `A` matrix, where A is $(X-\mu)'$. (Note the transpose)

Print this matrix below. It should have size 19×4.

In [2]:
```
# YOUR CODE GOES HERE
mu = np.mean(X, axis = 0)
A = (X-mu).T
print("M \n", mu)
print("A = \n", A)
```

```
M
 [ 0.75 -0.25  0.    0.5   0.5  -1.25 -1.    0.75  1.75  1.25 -0.25  0.5
  1.    1.25 -0.5   0.    1.5   0.75 -0.25]
A =
 [[-2.75  0.25  0.25  2.25]
 [ 1.25  2.25 -2.75 -0.75]
 [ 2.   -4.    2.    0.  ]
 [-3.5   1.5   0.5   1.5 ]
 [ 3.5  -4.5  -0.5   1.5 ]
 [ 2.25  3.25 -1.75 -3.75]
 [ 1.    6.   -4.   -3.  ]
 [ 2.25  1.25 -1.75 -1.75]
 [-1.75  0.25  1.25  0.25]
 [ 0.75 -0.25  1.75 -2.25]
 [ 1.25 -2.75 -1.75  3.25]
 [ 0.5  -0.5  -3.5   3.5 ]
 [ 1.   -1.   -3.    3.  ]
 [ 1.75 -0.25 -2.25  0.75]
 [-1.5  -1.5   1.5   1.5 ]
 [-3.    1.    0.    2.  ]
 [ 0.5  -0.5   3.5  -3.5 ]
 [ 0.25 -3.75  3.25  0.25]
 [ 0.25 -1.75  2.25 -0.75]]
```

"Small" covariance matrix

By transposing $X-\mu$ to get A, now we can compute a smaller covariance matrix with $A' A$. Compute this matrix, `C`, below and print the result.

In [3]:

```
# YOUR CODE GOES HERE
C = A.T @ A / X.shape[0]
print("C = \n", C)
```

```
C =
 [[ 17.46875  -4.71875  -6.59375  -6.15625]
 [ -4.71875  30.34375 -13.28125 -12.34375]
 [ -6.59375 -13.28125  24.59375  -4.71875]
 [ -6.15625 -12.34375  -4.71875  23.21875]]
```

Finding nonzero eigenvectors

Next, find the useful (nonzero) eigenvectors of `C`.

For validation purposes, there should be 3 useful eigenvectors, and the first one is `[-0.06628148 -0.79038331 0.47285044 0.38381435]`.

Keep these eigenvectors in a 4×3 array `e`.

In [4]:
```
# YOUR CODE GOES HERE
eigenvalues, eigenvectors = np.linalg.eig(C)
w= np.real(eigenvalues)
indices = np.argsort(-w)
e = eigenvectors[:,indices[0:3]]
print("Eigenvectors are:\n",e)
```

```
Eigenvectors are:
 [[-0.06628148  0.04124587 -0.86249959]
 [-0.79038331 -0.06822502  0.34733208]
 [ 0.47285044 -0.69123739  0.22046165]
 [ 0.38381435  0.71821654  0.29470586]]
```

Calculating "eigenfaces"

Now, we have all we need to compute `U`, the matrix of eigenfaces.

$\bf{U}_i = \bf{A} \bf{e}_i$

$(19 \times 3) = (19 \times 4)(4 \times 3)$

Compute and print U. Be sure to normalize your eigenvectors `e` before using the above equation.

In [5]:
```
# YOUR CODE GOES HERE

e_f = e/np.linalg.norm(e, axis = 0)
U_c = A @ e_f
U = U_c/np.linalg.norm(U_c, axis = 0)
print("Eigenfaces, U:\n",U)
```

```
Eigenfaces, U:
 [[ 0.07294372  0.12277459  0.33008441]
 [-0.26034151  0.11787331 -0.11677714]
 [ 0.29998485 -0.09606164 -0.27776956]
 [-0.01067529  0.04536213  0.42516696]
 [ 0.27653993  0.17530224 -0.44157072]
 [-0.37621372 -0.15082188 -0.23925816]
 [-0.59257956  0.02265222 -0.05657115]
 [-0.19897063 -0.0037123  -0.250194  ]
 [ 0.04569305 -0.07236581  0.20213547]
 [ 0.0084373  -0.25979087 -0.10504274]
 [ 0.18948616  0.35382298 -0.1518308 ]
 [ 0.00380575  0.46650428 -0.03585222]
 [ 0.03449119  0.40571147 -0.10256065]
 [-0.05241297  0.20419008 -0.19442141]
 [ 0.19396809  0.00756997  0.16057937]
 [ 0.01329023  0.11639359  0.36617258]
 [ 0.0508452  -0.45626561 -0.08985059]
 [ 0.3456779  -0.16842745 -0.07563409]
 [ 0.16171488 -0.18371276 -0.0569842 ]]
```

Projecting data into 3D

Now project your data into 3 dimensions with the formula:

$\Omega = U^\text{T} A $

$(3 \times 4) = (3 \times 19)(19 \times 4)$

Call the projected data Ω "`W`". Print `W.T`

In [6]:
```
# YOUR CODE GOES HERE
W = U.T @ A
print('Projected data in 3 dimensions:\n',W.T)
```

```
Projected data in 3 dimensions:
 [[ -0.8782013    0.44099733  -8.3011616 ]
 [-10.47224127  -0.72945617   3.34291139]
 [  6.26506632  -7.39065157   2.12184196]
 [  5.08537624   7.67911041   2.83640825]]
```

Reconstructing data in 19-D

We can project the transformed data `W` back into the original 19-D space using:

$\Gamma_f = U \Omega + \Psi$

where:

$\Gamma_f = $ reconstructed data

$U = $ eigenfaces

$\Omega = $ Reduced data

$\Psi = $ Means

Do this, and compute the MSE between each reconstructed sample and corresponding original points. Report all 4 MSE values.

In [7]:
```
# YOUR CODE GOES HERE
T_f = (U @ W).T +mu
MSE = np.mean((X - T_f)**2, axis = 1)
for i in range(4):
    print("MSE for sample %d: %e" %(i+1,MSE[i]))
```

```
MSE for sample 1: 4.513893e-30
MSE for sample 2: 3.537062e-30
MSE for sample 3: 5.950784e-30
MSE for sample 4: 1.128840e-30
```

2-D Reconstruction

What if we had only used the first 2 eigenvectors to compute the eigenfaces? Below, redo the earlier calculations, but use only two eigenfaces. Compute the 4 MSE values that you would get in this case.

(You should get an MSE of 3.626 for the first sample.)

In [8]:
```
# YOUR CODE GOES HERE

eigenvalues, eigenvectors = np.linalg.eig(C)
w= np.real(eigenvalues)
indices = np.argsort(-w)
e = eigenvectors[:,indices[0:2]]

e_f = e/np.linalg.norm(e, axis = 0)
U_c = A @ e_f
U = U_c/np.linalg.norm(U_c, axis = 0)
```

```
W = U.T @ A

T_f = U@W + mu.reshape(-1,1)
MSE2 = np.mean((X - T_f.T)**2, axis = 1)
print("Using only 2 eigenvectors:")
for i in range(4):
    print("MSE for sample %d: %e" %(i+1,MSE2[i]))
```

```
Using only 2 eigenvectors:
MSE for sample 1: 3.626804e+00
MSE for sample 2: 5.881609e-01
MSE for sample 3: 2.369586e-01
MSE for sample 4: 4.234322e-01
```

In [ ]:

```
```